CHAPTER 8

REDUCTIONS AND COMPLETENESS

Certain problems capture the difficulty of a whole complexity class. Logic plays a central role in this fascinating phenomenon.

8.1 REDUCTIONS

Like all complexity classes, **NP** contains an infinity of languages. Of the problems and languages we have seen so far in this book, **NP** contains TSP (D) (recall Section 1.3) and the SAT problem for Boolean expressions (recall Section 4.3). In addition, **NP** certainly contains REACHABILITY, defined in Section 1.1, and CIRCUIT VALUE from Section 4.3 (both are in **P**, and thus certainly in **NP**). It is intuitively clear, however, that the former two problems are somehow more worthy representatives of **NP** than the latter two. They seem to capture more faithfully the power and complexity of **NP**, they are not known (or believed) to be in **P** like the other two. We shall now introduce concepts that make this intuition precise and mathematically provable.

What we need is a precise notion of what it means for a problem to be *at least as hard as* another. We propose *reduction* (recall the discussion in Sections 1.2 and 3.2) as this concept. That is, we shall be prepared to say that problem A is at least as hard as problem B if B reduces to A. Recall what "reduces" means. We say that B reduces to A if there is a transformation R which, for every input x of B, produces an equivalent input $R(x)$ of A. Here by "equivalent" we mean that the answer to $R(x)$ considered as an input for A, "yes" or "no," is a correct answer to x, considered as an input of B. In other words, to solve B on input x we just have to compute $R(x)$ and solve A on it (see Figure 8.1).

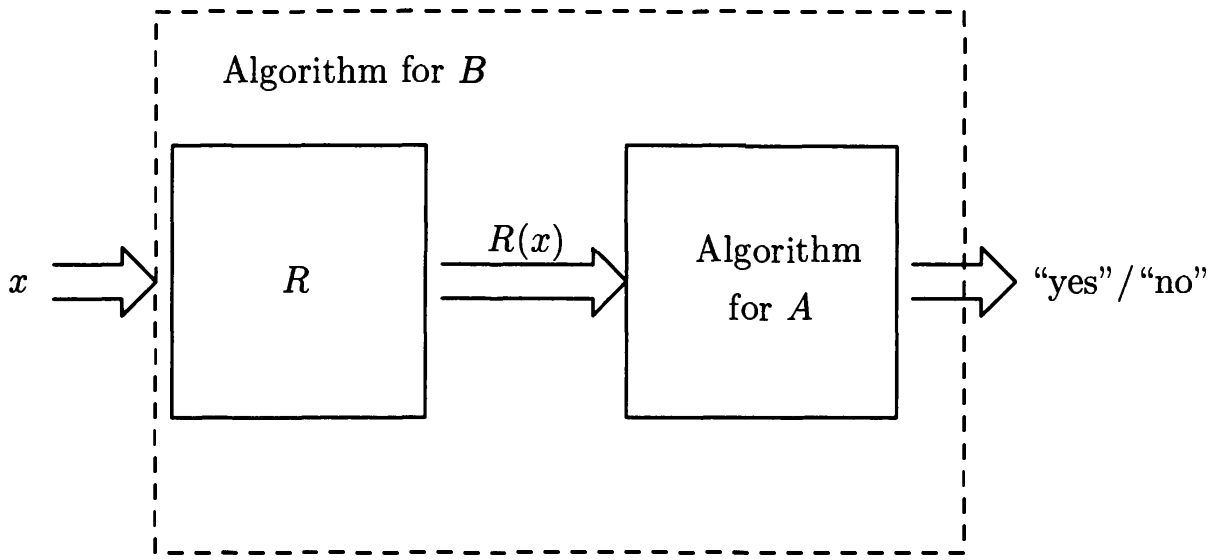

Figure 8-1. Reduction from B to A.

If the scenario in Figure 8.1 is possible, it seems reasonable to say that A is at least as hard as B. With one proviso: *That R should not be fantastically hard to compute.* If we do not limit the complexity of computing R, we could arrive at absurdities such as TSP (D) reduced to REACHABILITY, and thus REACHABILITY being harder than TSP (D)! Indeed, given any instance x of TSP (D) (that is, a distance matrix and a budget), we can apply the following reduction: Examine all tours; if one of them is cheaper than the budget, then $R(x)$ is the two-node graph consisting of a single edge from 1 to 2. Otherwise, it is the two-node graph with no edges. Notice that, indeed, $R(x)$ is a "yes" instance of REACHABILITY if and only if x was a "yes" instance of TSP (D). The flaw is, of course, that R is an exponential-time algorithm.

Definition 8.1: As we pointed out above, for our concept of reduction to be meaningful, it should involve the weakest computation possible. *We shall adopt $\log n$ space-bounded reduction as our notion of "efficient reduction."* That is, we say that language L_1 *is reducible to* L_2 if there is a function R from strings to strings computable by a deterministic Turing machine in space $\mathcal{O}(\log n)$ such that for all inputs x the following is true: $x \in L_1$ if and only if $R(x) \in L_2$. R is called *a reduction* from L_1 to L_2. □

Since our focal problems in complexity involve the comparisons of time classes, it is important to note that reductions are *polynomial-time algorithms.*

Proposition 8.1: If R is a reduction computed by Turing machine M, then for all inputs x M halts after a polynomial number of steps.

Proof: There are $\mathcal{O}(nc^{\log n})$ possible configurations for M on input x, where $n = |x|$. Since the machine is deterministic, no configuration can be repeated in the computation (because such repetition means the machine does not halt).

Thus, the computation is of length at most $\mathcal{O}(n^k)$ for some k. □

Needless to say, since the output string $R(x)$ is computed in polynomial time, its length is also polynomial (since at most one new symbol can be output at each step). We next see several interesting examples of reductions.

Example 8.1: Recall the problem HAMILTON PATH briefly discussed in Example 5.12. It asks, given a graph, whether there is a path that visits each node exactly once. Although HAMILTON PATH is a very hard problem, we next show that SAT (the problem of telling whether a given Boolean expression has a satisfying truth assignment) is at least as hard: We show that HAMILTON PATH can be reduced to SAT. We describe the reduction next.

Suppose that we are given a graph G. We shall construct a Boolean expression $R(G)$ such that $R(G)$ is satisfiable if and only if G has a Hamilton path. Suppose that G has n nodes, $1, 2, \ldots, n$. Then $R(G)$ will have n^2 Boolean variables, $x_{ij} : 1 \leq i, j, \leq n$. Informally, variable x_{ij} will represent the fact "node j is the ith node in the Hamilton path," which of course may be either true or false. $R(G)$ will be in conjunctive normal form, so we shall describe its clauses. The clauses will spell out all requirements on the x_{ij}'s that are sufficient to guarantee that they encode a true Hamilton path. To start, node j must appear in the path; this is captured by the clause $(x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj})$; we have such a clause for each j. But node j cannot appear both ith and kth: This is expressed by clause $(\neg x_{ij} \vee \neg x_{kj})$, repeated for all values of j, and $i \neq k$. Conversely, some node must be ith, thus we add the clause $(x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$ for each i; and no two nodes should be ith, or $(\neg x_{ij} \vee \neg x_{ik})$ for all i, and all $j \neq k$. Finally, for each pair (i, j) which is *not* an edge of G, it must not be the case that j comes right after i in the Hamilton path; therefore the following clauses are added for each pair (i, j) not in G and for $k = 1, \ldots, n-1$: $(\neg x_{ki} \vee \neg x_{k+1,j})$. This completes the construction. Expression $R(G)$ is the conjunction of all these clauses.

We claim that R is a reduction from HAMILTON PATH to SAT. To prove our claim, we have to establish two things: That for any graph G, expression $R(G)$ has a satisfying truth assignment if and only if G has a Hamilton path; and that R can be computed in space $\log n$.

Suppose that $R(G)$ has a satisfying truth assignment T. Since T satisfies all clauses of $R(G)$, it must be the case that, for each j there exists a unique i such that $T(x_{ij}) = \textbf{true}$, otherwise the clauses of the form $(x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj})$ and $(\neg x_{ij} \vee \neg x_{kj})$ cannot all be satisfied. Similarly, clauses $(x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$ and $(\neg x_{ij} \vee \neg x_{ik})$ guarantee that for each i there exists a unique j such that $T(x_{ij}) = \textbf{true}$. Hence, T really represents a *permutation* $\pi(1), \ldots, \pi(n)$ of the nodes of G, where $\pi(i) = j$ if and only if $T(x_{ij}) = \textbf{true}$. However, clauses $(\neg x_{k,i} \vee \neg x_{k+1,j})$ where (i, j) is not an edge of G and $k = 1, \ldots, n-1$ guarantee that, for all k, $(\pi(k), \pi(k+1))$ is an edge of G. This means that $(\pi(1), \pi(2), \ldots, \pi(n))$ is a

Hamilton path of G.

Conversely, suppose that G has a Hamilton path $(\pi(1), \pi(2), \ldots, \pi(n))$, where π is a permutation. Then it is clear that the truth assignment $T(x_{ij}) =$ **true** if $\pi(i) = j$, and $T(x_{ij}) =$ **false** if $\pi(i) \neq j$, satisfies all clauses of $R(G)$.

We still have to show that R can be computed in space $\log n$. Given G as an input, a Turing machine M outputs $R(G)$ as follows: First it writes n, the number of nodes of G, in binary, and, based on n it generates in its output tape, one by one, the clauses that do not depend on the graph (the first four groups in the description of $R(G)$). To this end, M just needs three counters, i, j, and k, to help construct the indices of the variables in the clauses. For the last group, the one that depends on G, M again generates one by one in its work string all clauses of the form $(\neg x_{ki} \vee \neg x_{k+1,j})$ for $k = 1, \ldots, n-1$; after such a clause is generated, M looks at its input to see whether (i, j) is an edge of G, and if it is not, then it outputs the clause. This completes our proof that HAMILTON PATH can be reduced to SAT.

We shall see many more reductions in this book; however, the present reduction is one of the simplest and clearest that we shall encounter. The clauses produced express in a straightforward and natural way the requirements of HAMILTON PATH, and the proof need only check that this translation is indeed accurate. Since SAT, the "target" problem, is a problem inspired from logic, one should not be surprised that it can "express" other problems quite readily: After all, expressiveness is logic's strongest suit. □

Example 8.2: We can also reduce REACHABILITY to CIRCUIT VALUE (another problem inspired by logic). We are given a graph G and wish to construct a variable-free circuit $R(G)$ such that the output of $R(G)$ is **true** if and only if there is a path from node 1 to node n in G.

The gates of $R(G)$ are of the form g_{ijk} with $1 \leq i, j \leq n$ and $0 \leq k \leq n$, and h_{ijk} with $1 \leq i, j, k \leq n$. Intuitively, g_{ijk} is **true** if and only if there is a path in G from node i to node j *not using any intermediate node bigger than* k. On the other hand, h_{ijk} will be **true** if and only if there is a path in G from node i to node j again not using intermediate nodes bigger than k, *but using* k *as an intermediate node.* We shall next describe each gate's sort and predecessors. For $k = 0$, all g_{ij0} gates are input gates (recall that there are no h_{ij0} gates). In particular, g_{ij0} is a **true** gate if and only if either $i = j$ or (i, j) is an edge of G, and it is a **false** gate otherwise. This is how the structure of G is reflected in $R(G)$. For $k = 1, \ldots, n$, h_{ijk} is an AND gate (that is, $s(h_{ijk}) = \wedge$), and its predecessors are $g_{i,k,k-1}$ and $g_{k,j,k-1}$ (that is, there are edges $(g_{i,k,k-1}, h_{ijk})$ and $(g_{k,j,k-1}, h_{ijk})$ in $R(G)$). Also, for $k = 1, \ldots, n$, g_{ijk} is an OR gate, and there are edges $(g_{i,j,k-1}, g_{ijk})$ and (h_{ijk}, g_{ijk}) in $R(G)$. Finally, g_{1nn} is the output gate. This completes the description of circuit $R(G)$.

It is easy to see that $R(G)$ is indeed a legitimate variable-free circuit, whose

gates can be renamed $1, 2, \ldots, 2n^3 + n^2$ (in nondecreasing order of the third index, say) so that edges go from lower-numbered gates to higher-numbered ones, and indegrees are in accordance to sorts. (Notice that there are no NOT gates in $R(G)$.) We shall next show that the value of the output gate of $R(G)$ is **true** if and only if there is a path from 1 to n in G.

We shall prove by induction on k that the values of the gates are indeed the informal meanings described above. The claim is true when $k = 0$, and if it is true up to $k - 1$, the definitions of h_{ijk} as $(g_{i,k,k-1} \wedge g_{k,j,k-1})$ and of h_{ijk} as $(h_{ijk} \vee g_{i,j;k-1})$ guarantee it to be true for k as well. Hence, the output gate g_{1nn} is **true** if and only if there is a path from 1 to n using no intermediate nodes numbered above n (of which there is none), that is, if and only if there is a path from 1 to n in G.

Furthermore, R can be computed in $\log n$ space. The machine would again go over all possible indices i, j, and k, and output the appropriate edges and sorts for the variables. Hence R is a reduction from REACHABILITY to CIRCUIT VALUE.

It is instructive to notice that circuit $R(G)$ is derived from a polynomial-time algorithm for REACHABILITY, namely the well-known Floyd-Warshall algorithm. As we shall see soon, rendering polynomial algorithms as variable-free circuits is a quite general pattern. It is remarkable that the circuit uses no NOT gates, and is thus a *monotone circuit* (see Problems 4.4.13 and 8.4.7). Finally, notice that the circuit constructed has *depth* (length of the longest path from an input to an output gate) that is linear in n. In Chapter 15 we shall exhibit a much "shallower" circuit for the same problem. □

Example 8.3: We can also reduce CIRCUIT SAT (recall Section 4.3) to SAT. We are given a circuit C, and wish to produce a Boolean expression $R(C)$ such that $R(C)$ is satisfiable if and only if C is satisfiable. But this is not hard to do, since expressions and circuits are different ways of representing Boolean functions, and translations back and forth are easy. The variables of $R(C)$ will contain all variables appearing in C, and in addition, for each gate g of C we are going to have a variable in $R(C)$, also denoted g. For each gate of C we shall generate certain clauses of $R(C)$. If g is a variable gate, say corresponding to variable x, then we add the two clauses $(\neg g \vee x)$ and $(g \vee \neg x)$. Notice that any truth assignment T that satisfies both clauses must have $T(g) = T(x)$; to put it otherwise, $(\neg g \vee x) \wedge (g \vee \neg x)$ is the conjunctive normal form of $g \Leftrightarrow x$. If g is a **true** gate, then we add the clause (g); if it is a **false** gate, we add the clause $(\neg g)$. If g is a NOT gate, and its predecessor in C is gate h, we add the gates $(\neg g \vee \neg h)$ and $(g \vee h)$ (the conjunctive normal form of $(g \Leftrightarrow \neg h)$). If g is an OR gate with predecessors h and h', then we add to $R(C)$ the clauses $(\neg h \vee g)$, $(\neg h' \vee g)$, and $(h \vee h' \vee \neg g)$ (the conjunctive normal form of $g \Leftrightarrow (h \vee h')$). Similarly, if g is an AND gate with predecessors h and h', then we add to $R(C)$

the clauses $(\neg g \vee h)$, $(\neg g \vee h')$, and $(\neg h \vee \neg h' \vee g)$. Finally, if g is also the output gate, we add to $R(C)$ the clause (g). It is easy to see that $R(C)$ is satisfiable if and only if C was, and that the construction can be carried out within $\log n$ space. □

Example 8.4: One trivial but very useful kind of reduction is *reduction by generalization*. We say, informally, that problem A is a *special case* of problem B if the inputs of A comprise an easily recognizable subset of the inputs of B, and on those inputs A and B have the same answers. For example, CIRCUIT VALUE is a special case of CIRCUIT SAT: Its inputs are all circuits that happen to be variable-free; and on those circuits the CIRCUIT VALUE problem and the CIRCUIT SAT problem have identical answers. Another way to say the same thing is that CIRCUIT SAT is a *generalization* of CIRCUIT VALUE. Notice that there is a trivial reduction from CIRCUIT VALUE to CIRCUIT SAT: Just take R to be the identity function. □

There is a chain of reductions that can be traced in the above examples: From REACHABILITY to CIRCUIT VALUE, to CIRCUIT SAT, to SAT. Do we then have a reduction from REACHABILITY to SAT? That reductions compose requires some proof:

Proposition 8.2: If R is a reduction from language L_1 to L_2 and R' is a reduction from language L_2 to L_3, then the composition $R \cdot R'$ is a reduction from L_1 to L_3.

Proof: That $x \in L_1$ if and only if $R'(R(x)) \in L_3$ is immediate from the fact that R and R' are reductions. The nontrivial part is to show that $R \cdot R'$ can be computed in space $\log n$.

One first idea is to compose the two machines with input and output, M_R and $M_{R'}$, that compute R and R' respectively (Figure 8.2) so that $R(x)$ is first produced, and from it the final output $R'(R(x))$. Alas, the composite machine M must have $R(x)$ written on a work string; and $R(x)$ may be much longer than $\log |x|$.

The solution to this problem is clever and simple: We do not explicitly store the intermediate result in a string of M. Instead, we simulate $M_{R'}$ on input $R(x)$ by remembering at all times the cursor position i of the input string of $M_{R'}$ (which is the output string of M_R). i is stored in binary in a new string of M. Initially $i = 1$, and we have a separate set of strings on which we are about to begin the simulation of M_R on input x.

Since we know that the input cursor in the beginning scans a $\triangleright$, it is easy to simulate the first move of $M_{R'}$. Whenever the cursor of $M_{R'}$'s input string moves to the right, we increment i by one, and continue the computation of machine M_R on input x (on the separate set of strings) long enough for it to produce the next output symbol; this is the symbol currently scanned by the input cursor of $M_{R'}$, and so the simulation can go on. If the cursor stays at the

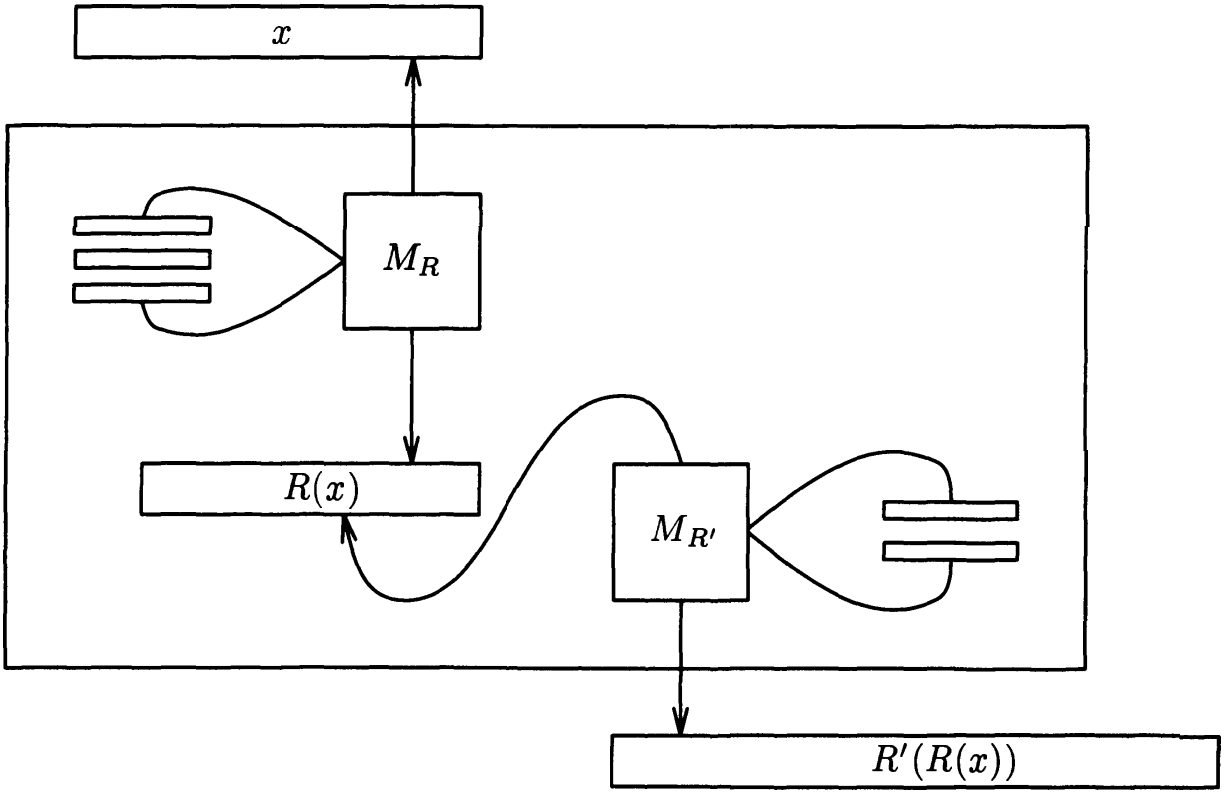

Figure 8-2. How *not* to compose reductions.

same position, we just remember the input symbol scanned. If, however, the input cursor of $M_{R'}$ moves to the left, there is no obvious way to continue the simulation, since the previous symbol output by M_R has been long forgotten. We must do something more radical: We decrement i by one, and then run M_R on x *from the beginning*, counting on a separate string the symbols output, and stopping when the ith symbol is output. Once we know this symbol, the simulation of $M_{R'}$ can be resumed. It is clear that this machine indeed computes $R \cdot R'$ in space $\log n$ (recall that $|R(x)|$ is at most polynomial in $n = |x|$, and so i has $\mathcal{O}(\log n)$ bits). □

8.2 COMPLETENESS

Since reducibility is transitive, it orders problems with respect to their difficulty. We shall be particularly interested in the *maximal elements* of this partial order:

Definition 8.2: Let $\mathcal{C}$ be a complexity class, and let L be a language in $\mathcal{C}$. We say that L is *$\mathcal{C}$-complete* if any language $L' \in \mathcal{C}$ can be reduced to L. □

Although it is not *a priori* clear that complete problems exist, we shall soon show that certain natural and familiar problems are **NP**-complete, and others **P**-complete; in future chapters we shall introduce **PSPACE**-complete problems, **NL**-complete ones, and more.

Complete problems comprise an extremely central concept and method-

ological tool for complexity theory (with **NP**-complete problems perhaps the best-known example). We feel that we have completely understood and categorized the complexity of a problem only if the problem is known to be complete for its complexity class. On the other hand, complete problems capture the essence and difficulty of a class. They are the link that keeps complexity classes alive and anchored in computational practice. For example, the existence of important, natural problems that are complete for a class lends the class a significance that may not be immediately clear from its definition (**NP** is a case in point). Conversely, the absence of natural complete problems makes a class suspect of being artificial and superfluous. However, the most common use of completeness is as a *negative complexity result:* A complete problem is the least likely among all problems in $\mathcal{C}$ to belong in a weaker class $\mathcal{C}' \subseteq \mathcal{C}$; if it does, then the whole class $\mathcal{C}$ coincides with the weaker class $\mathcal{C}'$ —*as long as* $\mathcal{C}'$ *is closed under reductions.* We say that a class $\mathcal{C}'$ is closed under reductions if, whenever L is reducible to L' and $L' \in \mathcal{C}'$, then also $L \in \mathcal{C}'$. All classes of interest are of this kind:

Proposition 8.3: **P**, **NP**, **coNP**, **L**, **NL**, **PSPACE**, and **EXP** are all closed under reductions.

Proof: Problem 8.4.3. □

Hence, if a **P**-complete problem is in **L**, then **P** = **L**, and if it is in **NL** then **P** = **NL**. If an **NP**-complete problem is in **P**, then **P** = **NP**. And so on. In this sense, *complete problems are a valuable tool for showing complexity classes coincide:*

Proposition 8.4: If two classes $\mathcal{C}$ and $\mathcal{C}'$ are both closed under reductions, and there is a language L which is complete for both $\mathcal{C}$ and $\mathcal{C}'$, then $\mathcal{C} = \mathcal{C}'$.

Proof: Since L is complete for $\mathcal{C}$, all languages in $\mathcal{C}$ reduce to $L \in \mathcal{C}'$. Since $\mathcal{C}'$ is closed under reductions, it follows that $\mathcal{C} \subseteq \mathcal{C}'$. The other inclusion follows by symmetry. □

This result is one aspect of the usefulness of complete problems in the study of complexity. We shall use this method of class identification several times in Chapters 16, 19, and 20.

To exhibit our first **P**-complete and **NP**-complete problems, we employ a useful method for understanding time complexity which could be called the *table method* (recall the reachability method for space complexity). Consider a polynomial-time Turing machine $M = (K, \Sigma, \delta, s)$ deciding language L. Its computation on input x can be thought of as a $|x|^k \times |x|^k$ *computation table* (see Figure 8.3), where $|x|^k$ is the time bound. In this table rows are time steps (ranging from 0 to $|x|^k - 1$), while columns are positions in the string of the machine (the same range). Thus, the (i, j)th table entry represents the contents of position j of the string of M at time i (i.e., after i steps of the machine).

$\triangleright$	0_s	1	1	0	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1_{q_0}	1	0	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1_{q_0}	0	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1	0_{q_0}	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1	0	$\sqcup_{q_0}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1	$0_{q_0'}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1_q	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1_q	1	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright_q$	1	1	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1_s	1	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	1_{q_1}	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	1	$\sqcup_{q_1}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	$1_{q_1'}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright_q$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	$\sqcup_s$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	"yes"	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$

Figure 8.3. Computation table.

We shall standardize the computation table a little, so that it is more simple and flexible. Since we know that any k-string Turing machine can be simulated by a single-string machine within a polynomial, it is not a loss of generality to assume that M has one string, and that on any input x it halts after at most $|x|^k - 2$ steps (we can take k high enough so that this holds for $x \geq 2$, and we ignore what happens when $|x| \leq 1$). The computation table pads the string with enough $\sqcup$'s to its right end so that the total length is $|x|^k$; since Turing machine strings are extended during the computation by $\sqcup$'s, this padding of the computation table is no departure from our conventions. Notice that the actual computation will never get to the right end of the table, for lack of time. If at time i the state is q and the cursor scans the jth position, then the (i,j)th entry of the table is not just the symbol σ contained at position j at time i, but a new symbol σ_q, and thus cursor position and state are also recorded nicely. However, if q above is "yes" or "no", then instead of the symbol σ_q we simply have "yes" or "no" as an entry of the table.

We further modify the machine so that the cursor starts not at $\triangleright$, but at the first symbol of the input. Also, the cursor never visits the leftmost $\triangleright$; since such a visit would be followed immediately by a right move, this is achieved by telescoping two moves of the machine each time the cursor is about to move to the leftmost $\triangleright$. Thus, the first symbol in every row of the computation table is a $\triangleright$ (and never an $\triangleright_q$). Finally, we shall assume that, if the machine has halted

before its time bound of n^k has expired, and thus one of the symbols "yes" or "no" appear at a row before the last, then all subsequent rows will be identical to that one. We say that the table is *accepting* if $T_{|x|^k-1,j}$ = "yes" for some j.

Example 8.5: Recall the machine of Example 2.3 deciding palindromes within time n^2. In Figure 8.3 we show its computation table for input 0110. It is an accepting table. □

The folowing result follows immediately from the definition of the computation table:

Proposition 8.5: M accepts x if and only if the computation table of M on input x is accepting. □

We are now ready for our first completeness result:

Theorem 8.1: CIRCUIT VALUE is **P**-complete.

Proof: We know that CIRCUIT VALUE is in **P** (this is a prerequisite for a problem to be **P**-complete, recall Definition 8.2). We shall show that for any language $L \in \mathbf{P}$ there is a reduction R from L to CIRCUIT VALUE.

Given any input x, $R(x)$ must be a variable-free circuit such that $x \in L$ if and only if the value of $R(x)$ is **true**. Let M be the Turing machine that decides L in time n^k, and consider the computation table of M on x, call it T. When $i = 0$, or $j = 0$, or $j = |x|^k - 1$, then the value of T_{ij} is *a priori* known (the jth symbol of x or a ⊔ in the first case, a ▷ in the second, a ⊔ in the third).

Consider now any other entry T_{ij} of the table. The value of T_{ij} reflects the contents of position j of the string at time i, *which depends only on the contents of the same position or adjacent positions at time* $i-1$. That is, T_{ij} depends only on the entries $T_{i-1,j-1}$, $T_{i-1,j}$, and $T_{i-1,j+1}$ (see Figure 8.4(a)). For example, if all three entries are symbols in Σ, then this means that the cursor at step i is not at or around position j of the string, and hence T_{ij} is the same as $T_{i-1,j}$. If one of the entries $T_{i-1,j-1}$, $T_{i-1,j}$, or $T_{i-1,j+1}$ is of the form σ_q, then T_{ij} may be a new symbol written at step i, or of the form σ_q if the cursor moves to position j, or perhaps again the same symbol as $T_{i-1,j}$. *In all cases, to determine* T_{ij} *we need only look at* $T_{i-1,j-1}$, $T_{i-1,j}$, *and* $T_{i-1,j+1}$.

Let Γ denote the set of all symbols that can appear on the table (symbols of the alphabet of M, or symbol-state combinations). Encode next each symbol $\sigma \in \Gamma$ as a vector $(s_1, \ldots, s_m)$, where $s_1, \ldots, s_m \in \{0,1\}$, and $m = \lceil \log |\Gamma| \rceil$. The computation table can now be thought of as a table of binary entries $S_{ij\ell}$ with $0 \leq i \leq n^k - 1$, $0 \leq j \leq n^k - 1$, and $1 \leq \ell \leq m$. By the observation in the previous paragraph, each binary entry $S_{ij\ell}$ only depends on the $3m$ entries $S_{i-1,j-1,\ell'}$, $S_{i-1,j,\ell'}$, and $S_{i-1,j+1,\ell'}$, where ℓ' ranges over $1, \ldots, m$. That is, *there are* m *Boolean functions* $F_1, \ldots, F_m$ with $3m$ inputs each such that, for all $i, j > 0$

$$S_{ij\ell} = F_\ell(S_{i-1,j-1,1}, \ldots, S_{i-1,j-1,m}, S_{i-1,j,1}, \ldots, S_{i-1,j+1,m})$$

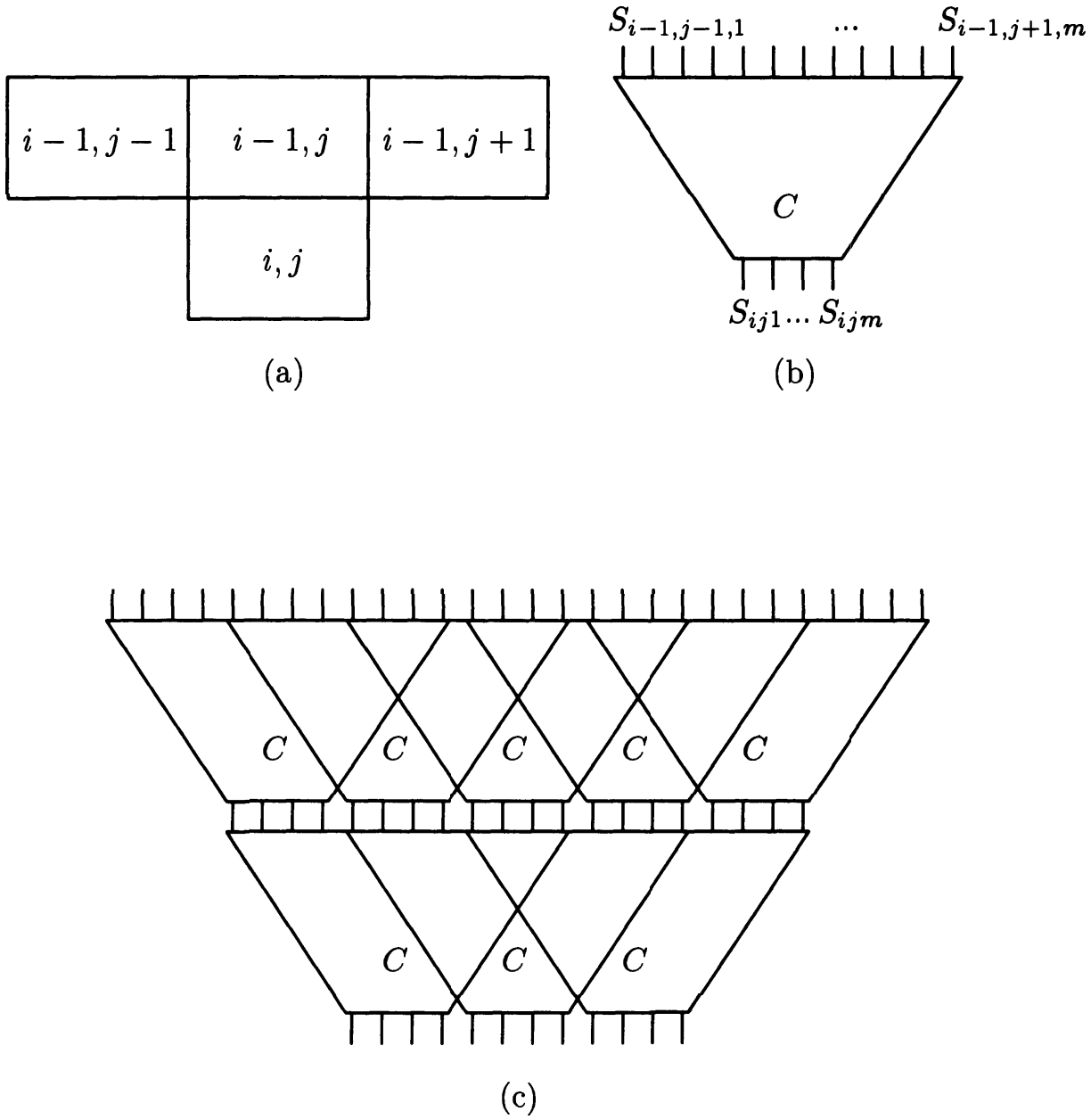

Figure 8-4. The construction of the circuit.

(we call these functions Boolean by disregarding for a moment the difference between **false-true** and 0-1). Since every Boolean function can be rendered as a Boolean circuit (recall Section 4.3), it follows that there is a Boolean circuit C with $3m$ inputs and m outputs that computes the binary encoding of T_{ij} given the binary encodings of $T_{i-1,j-1}$, $T_{i-1,j}$, and $T_{i-1,j+1}$ for all $i = 1, \ldots, |x|^k$ and $j = 1, \ldots, |x|^k - 1$ (see Figure 8.4(b)). Circuit C depends only on M, and has a fixed, constant size, independent of the length of x.

We are now ready to describe our reduction R from L, the language in **P** decided by M, to CIRCUIT VALUE. For each input x, $R(x)$ *will basically consist of* $(|x|^k - 1) \cdot (|x|^k - 2)$ *copies of the circuit* C (Figure 8.4(c)), one for each entry T_{ij} of the computation table that is not on the top row or the two extreme columns. Let us call C_{ij} the (i,j)th copy of C. For $i \geq 1$, the input gates of C_{ij} will be identified with the output gates of $C_{i-1,j-1}$, $C_{i-1,j}$, and $C_{i-1,j+1}$. The input gates of the overall circuit are the gates corresponding to the first row, and the first and last column. The sorts (**true** or **false**) of these gates correspond to the known contents of these three lines. Finally, the output gate of the $R(x)$ is the first output of circuit $C_{|x|^k-1,1}$ (here we are assuming, with no loss of generality, that M always ends with "yes" or "no" on its second string position, and that the first bit of the encoding of "yes" is 1, whereas the of "no" is 0). This completes the description of $R(x)$.

We claim that the value of circuit $R(x)$ is **true** if and only if $x \in L$. Suppose that the value of $R(x)$ is indeed **true**. It is easy to establish by induction on i that the values of the outputs of circuits C_{ij} spell in binary the entries of the computation table of M on x. Since the output of $R(x)$ is **true**, this means that entry $T_{|x|^k-1,1}$ of the computation table is "yes" (since it can only be "yes" or "no", and the encoding of "no" starts with a 0). It follows that the table is accepting, and thus M accepts x; therefore $x \in L$.

Conversely, if $x \in L$ then the computation table is accepting, and thus the value of the circuit $R(x)$ is **true**, as required.

It remains to argue that R can be carried out in $\log |x|$ space. Recall that circuit C is fixed, depending only on M. The computation of R entails constructing the input gates (easy to do by inspecting x and counting up to $|x|^k$), and generating many indexed copies of the fixed circuit C and identifying appropriate input and output gates of these copies—tasks involving straightforward manipulations of indices, and thus easy to perform in $\mathcal{O}(\log |x|)$ space. □

In CIRCUIT VALUE we allow AND, OR, and NOT gates in our circuits (besides the input gates, of course). As it turns out, *NOT gates can be eliminated*, and the problem remains **P**-complete. This is rather surprising, because it is well-known that circuits with only AND and OR gates are less expressive than general circuits: They can only compute *monotone Boolean functions* (recall Problem 4.4.13). Despite the fact that monotone circuits are far less expressive than general circuits, monotone circuits with constant inputs are as difficult to evaluate as general ones. To see this, notice that, given any general circuit we can "move the NOTs downwards," applying De Morgan's Laws at each step (basically, changing all ANDs to ORs and vice-versa) until they are applied to inputs. Then we can simply change $\neg$**true** to **false** (creating a new input gate each time), or vice-versa. This modification of the circuit can obviously be carried out in logarithmic space. We therefore have:

Corollary: MONOTONE CIRCUIT VALUE is **P**-complete.

For other special cases of CIRCUIT VALUE see Problems 8.4.7.

We next prove our first **NP**-completeness result, reusing much of the machinery developed for Theorem 8.1.

Theorem 8.2 (Cook's Theorem): SAT is **NP**-complete.

Proof: The problem is in **NP**: Given a satisfiable Boolean expression, a nondeterministic machine can "guess" the satisfying truth assignment and verify it in polynomial time. Since we know (Example 8.3) that CIRCUIT SAT reduces to SAT, we need to show that all languages in **NP** can be reduced to CIRCUIT SAT.

Let $L \in \mathbf{NP}$. We shall describe a reduction R which for each string x constructs a circuit $R(x)$ (with inputs that can be either variables or constants) such that $x \in L$ if and only if $R(x)$ is satisfiable. Since $L \in \mathbf{NP}$, there is a nondeterministic Turing machine $M = (K, \Sigma, \Delta, s)$ that decides L in time n^k. That is, given a string x, there is an accepting computation (sequence of nondeterministic choices) of M on input x if and only if $x \in L$. We assume that M has a single string; furthermore, we can assume that it has at each step *two nondeterministic choices*. If for some state-symbol combinations there are $m > 2$ choices in Δ, we modify M by adding $m-2$ new states so that the same effect is achieved (see Figure 8.5 for an illustration). If for some combination there is only one choice, we consider that the two choices coincide; and finally if for some state-symbol combination there is no choice in Δ, we add to Δ the choice that changes no state, symbol, or position. So, machine M has exactly two choices for each symbol-state combination. One of these choices is called choice 0 and the other choice 1, so that a sequence of nondeterministic choices is simply a bitstring $(c_1, c_2, \ldots, c_{|x|^k-1}) \in \{0,1\}^{|x|^{k-1}}$.

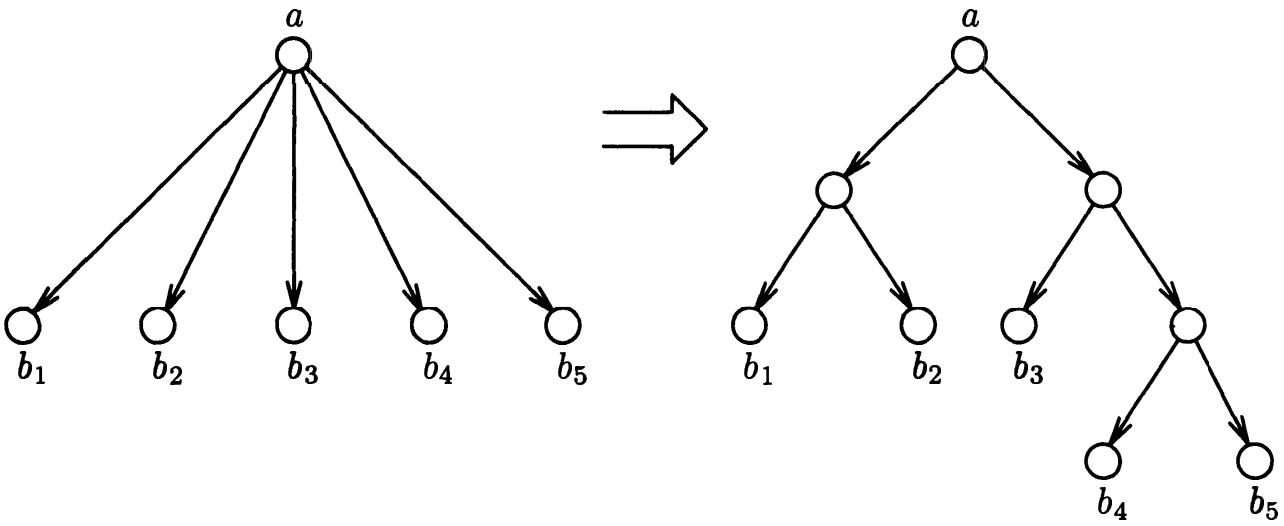

Figure 8-5. Reducing the degree of nondeterminism.

Since the computation of nondeterministic Turing machines proceeds in parallel paths (recall Figure 2.9), there is no simple notion of computation table that captures all of the behavior of such a machine on an input. *If, however, we fix a sequence of choices* $\mathbf{c} = (c_0, c_2, \ldots, c_{|x|^k-1})$, then the computation is effectively deterministic (at the ith step we take choice c_i), and thus we can define the computation table $T(M, x, \mathbf{c})$ corresponding to the machine, input, and sequence of choices. Again the top row and the extreme columns of the table will be predetermined. All other entries T_{ij} will depend only on the entries $T_{i-1,j-1}$, $T_{i-1,j}$, and $T_{i-1,j+1}$ *and the choice* c_{i-1} *at the previous step* (see Figure 8.6). That is, this time the fixed circuit C has $3m + 1$ entries instead of $3m$, the extra entry corresponding to the nondeterministic choice. Thus we can again construct in $\mathcal{O}(\log |x|)$ space a circuit $R(x)$, this time with variable gates $c_0, c_1, \ldots, c_{|x|^k-1}$ corresponding to the nondeterministic choices of the machine. It follows immediately that $R(x)$ is satisfiable (that is, there is a sequence of choices $c_0, c_1, \ldots, c_{|x|^k-1}$ such that the computation table is accepting) if and only if $x \in L$. □

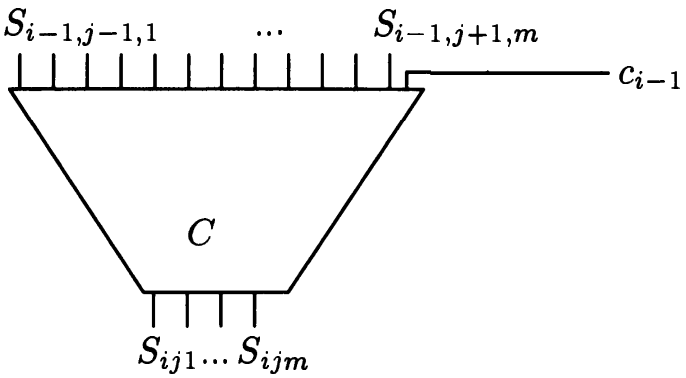

Figure 8-6. The construction for Cook's theorem.

We shall see many more **NP**-complete problems in the next chapter.

8.3 LOGICAL CHARACTERIZATIONS

Theorems 8.1 and 8.2, establishing that two computational problems from logic are complete for our two most important complexity classes, are ample evidence of the close relationship between logic and complexity. There is an interesting parallel, pursued in this section, in which logic captures complexity classes in an even more direct way. Recall that we can associate with each expression ϕ of first-order logic (or of existential second-order logic, recall Section 5.7) in the vocabulary of graph theory a computational problem called ϕ-GRAPHS, asking whether a given finite graph satisfies ϕ. In Section 5.7 we showed that for any expression ϕ in existential second-order logic ϕ-GRAPHS is in **NP**, and it is in **P** if ϕ is Horn. *We shall now show the converse statements.*

Let $\mathcal{G}$ be a set of finite graphs—that is, a graph-theoretic property. The computational problem corresponding to $\mathcal{G}$ is to decide, given a graph G, whether $G \in \mathcal{G}$. We say that $\mathcal{G}$ is *expressible in existential second-order logic* if there is an existential second-order logic sentence $\exists P\phi$ such that $G \models \exists P\phi$ if and only if $G \in \mathcal{G}$.

Naturally, there are many computational problems that do not correspond to properties of graphs. It can be argued, however, that this is an artifact of our preference for strings over graphs as the basis of our encodings. Graphs are perfectly adequate for encoding arbitrary mathematical objects. For example, any language L can be thought of as a set of graphs $\mathcal{G}$, where $G \in \mathcal{G}$ if and only if the first row of the adjacency matrix of G spells a string in L. With this in mind *(and only in this section)* we shall denote by **P** the sets of graph-theoretic properties whose corresponding computational problem is in **P**, and the same for **NP**†.

Theorem 8.3 (Fagin's Theorem): The class of all graph-theoretic properties expressible in existential second-order logic is precisely **NP**.

Proof: If $\mathcal{G}$ is expressible in existential second-order logic, we already know from Theorem 5.8 that it is indeed in **NP**. For the other direction, suppose that $\mathcal{G}$ is a graph property in **NP**. That is, there is a nondeterministic Turing machine M deciding whether $G \in \mathcal{G}$ for some graph G with n nodes in time n^k for some integer $k > 2$. We shall construct a second-order expression $\exists P\phi$ such that $G \models \exists P\phi$ if and only if $G \in \mathcal{G}$.

We must first standardize our nondeterministic machines a little more. We can assume that the input of M is the adjacency matrix of the graph under consideration. In fact, we shall assume that the adjacency matrix is spread over the input string in a rather peculiar way: The input starts off with the $(1,1)$st entry of the adjacency matrix, and between any two entries we have $n^{k-2} - 1$ ⊔'s. That is, the input is spread over n^k positions of the string; since the machine may start by condensing its input, this is no loss of generality.

We are now ready to start our description of $\exists P\phi$. P will be a relation symbol with very high arity. In fact, it will be much more clear to describe an equivalent expression of the form $\exists P_1 \ldots \exists P_m \phi$, where the P_i's are relational symbols; P will then be the Cartesian product of the P_i's. We call the P_i's the *new relations*, and describe them next.

First, S is a binary new relation symbol whose intention is to represent a *successor function over the nodes of G;* that is, in any model M of ϕ, S will be a relation isomorphic to $\{(0,1),(1,2),\ldots,(n-2,n-1)\}$ (notice that here we

---

† If the reader feels uncomfortable with this sudden change in our most basic conventions, there is another way of stating Theorem 8.3: **NP** is precisely the class of all languages that are *reducible* to some graph-theoretic property expressed in existential second-order logic; similarly for Theorem 8.4. See Problem 8.4.12.

assume that the nodes of G are $0, 1, \ldots, n-1$ instead of the usual $1, 2, \ldots, n$; this is, of course, inconsequential). We shall not describe now how S can be prescribed in first-order logic (Problem 8.4.11); most of the work has already been done in Example 5.12 where a Hamilton path was specified.

Once we have S, and thus we can identify the nodes of G with the integers $0, 1, \ldots, n-1$, we can define some interesting relations. For example, $\zeta(x)$ is an abbreviation of the expression $\forall y \neg S(y, x)$, which states that node x is 0, the element with no predecessor in S; on the other hand, $\eta(j)$ is an expression which abbreviates $\forall y \neg S(x, y)$, stating that node x equals $n-1$.

Since the variables stand for numbers between 0 and $n-1$ (whatever $n-1$ may be in the present model), k-tuples of variables may be used to represent numbers between 0 and $n^k - 1$ *with k the degree of the polynomial bound of M*. We shall abbreviate k-tuples of variables $(x_1, \ldots, x_k)$ as $\mathbf{x}$. In fact, we can define a first-order expression S_k with $2k$ free variables, such that $S_k(\mathbf{x}, \mathbf{y})$ if and only if $\mathbf{y}$ encodes the k-digit n-ary number that comes after the one encoded by $\mathbf{x}$. That is, S_k is the successor function in $\{0, 1, \ldots, n^k - 1\}$.

We shall define S_j inductively on j. First, if $j = 1$ then obviously S_1 is S itself. For the inductive step, suppose that we already have an expression $S_{j-1}(x_1, \ldots, x_{k-1}, y_1, \ldots, y_{j-1})$ that defines the successor function for $j-1$ digits. Then the expression defining S_j is this (universally quantified over all variables):

$$[S(x_j, y_j) \wedge (x_1 = y_1) \wedge \ldots \wedge (x_{j-1} = y_{j-1}))] \vee$$
$$[\eta(x_j) \wedge \zeta(y_j) \wedge S_{j-1}(x_1, \ldots, x_{j-1}, y_1, \ldots, y_{j-1})]$$

That is, in order to obtain from $x_1 \ldots x_j$ the n-ary description of its successor $y_1 \ldots y_j$ (least significant digit first) we do this: If the last digit of $\mathbf{x}$ is not $n-1$ (first line), then we just increment it and keep all other digits the same. But if it is $n-1$, then it becomes zero, and the remaining $j-1$-digit number is incremented, recursively. Thus, $S_k(\mathbf{x}, \mathbf{y})$ is indeed a first-order expression, involving $\mathcal{O}(k^2)$ symbols, satisfied if and only if $\mathbf{x}$ and $\mathbf{y}$ are consecutive integers between zero and $n^k - 1$. It will appear in several places in the expression $\exists P \phi$ described below.

Now that we have S_k, and therefore "we can count up to n^k," we can describe the computation table for M on input x. In particular, for each symbol σ appearing on the computation table, we have a $2k$-ary new relation symbol T_σ. $T_\sigma(\mathbf{x}, \mathbf{y})$ means that the (i, j)th entry of the computation table T is symbol σ, where $\mathbf{x}$ encodes i and $\mathbf{y}$ encodes j. Finally, for the two nondeterministic choices 0 and 1 at each step of M we have two k-ary symbols C_0 and C_1; for example, $C_0(\mathbf{x})$ means that at the ith step, where $\mathbf{x}$ encodes i, the 0th nondeterministic choice is made. These are all the new relations; the second-order formula will thus be of the form $\exists S \exists T_{\sigma_1} \ldots \exists T_{\sigma_k} \exists C_0 \exists C_1 \phi$.

All that remains is to describe ϕ. ϕ essentially states (besides the fact that S is a successor relation, which we omit) the following:

(a) The top row and the extreme columns of T are as they should be in a legal computation table of M on input x.
(b) All remaining entries are filled according to the transition relation of M.
(c) One nondeterministic choice is taken at each step. Finally,
(d) The machine ends accepting.

For part (a) we have to state that, if $\mathbf{x}$ encodes 0 then $T_{\sqcup}(\mathbf{x}, \mathbf{y})$, *unless* the last $k-2$ components of $\mathbf{y}$ are all 0, in which case $T_1(\mathbf{x}, \mathbf{y})$ or $T_0(\mathbf{x}, \mathbf{y})$, *depending on whether or not* $G(y_1, y_2)$, where G is the input graph (recall our peculiar input convention). This is the only place where G occurs in ϕ. Also for part (a) we must state that, if $\mathbf{y}$ encodes 0 then $T_{\triangleright}(\mathbf{x}, \mathbf{y})$, while if $\mathbf{y}$ encodes $n^k - 1$ then $T_{\sqcup}(\mathbf{x}, \mathbf{y})$

For part (b), we must require that the computation table reflects the transition relation of M. Notice that the transition relation of M can be expressed as a set of quintuples $(\alpha, \beta, \gamma, c, \sigma)$ where $\alpha, \beta, \gamma, \sigma$ are table symbols, and $c \in \{0, 1\}$ is a nondeterministic choice. Each such quintuple means that, whenever $T(i-1, j-1) = \alpha$, $T(i-1, j) = \beta$ $T(i-1, j+1) = \gamma$, and the choice c was made at the $i-1$st step, then $T(i, j) = \sigma$. For each such quintuple we have the following conjunct in ϕ:

$$\underline{[S_k(\mathbf{x}', \mathbf{x}) \wedge S_k(\mathbf{y}', \mathbf{y}) \wedge S_k(\mathbf{y}, \mathbf{y}'') \wedge T_\alpha(\mathbf{x}', \mathbf{y}') \wedge T_\beta(\mathbf{x}', \mathbf{y}) \wedge T_\gamma(\mathbf{x}', \mathbf{y}'') \wedge C_c(\mathbf{x}')] \Rightarrow T_\sigma(\mathbf{x}, \mathbf{y})}.$$

The appearances of S_k in this expression are independent copies of the expression defined inductively earlier in the proof.

Part (c) states that at all times exactly one of the nondeterministic choices is taken:

$$(C_0(\mathbf{x}) \vee C_1(\mathbf{x})) \wedge (\neg C_0(\mathbf{x}) \vee \neg C_1(\mathbf{x})). \tag{1}$$

Interestingly, this is a crucial part of the construction—for example, it is the only place where we have a non-Horn clause!

Part (d) is the easiest one: $\theta(\mathbf{x}, \mathbf{y}) \Rightarrow \neg T_{\text{“no”}}(\mathbf{x}, \mathbf{y})$ where $\theta(\mathbf{x}, \mathbf{y})$ abbreviates the obvious expression stating that $\mathbf{x}$ encodes $n^k - 1$ and $\mathbf{y}$ encodes 1 (recall our convention that the machine stops with "yes" or "no" at the first string position). The conjunction of all these clauses is then preceded by $5k$ universal quantifiers, corresponding to the variable groups $\mathbf{x}$, $\mathbf{x}'$, $\mathbf{y}$, $\mathbf{y}'$, $\mathbf{y}''$. The construction of the expression is now complete.

We claim that a given graph G satisfies the second-order expression above if and only if $G \in \mathcal{G}$. The expression was constructed in such a way that it is satisfied by exactly those graphs G which, when input to M, have an accepting sequence of nondeterministic choices, and thus an accepting computation table; that is, precisely the graphs in $\mathcal{G}$. □

We would have liked to conclude this section with the converse of Theorem 5.9, stating that the set of properties expressible in Horn existential second-order expressions is precisely **P**. Unfortunately, *this is not true.* There are certain computationally trivial graph-theoretic properties such as "the graph has an even number of edges" which cannot be expressed in Horn existential second-order logic (see Problems 8.4.15). The difficulty is of a rather unexpected nature: Of all the ingredients needed to express deterministic polynomial computation (in the style of the previous proof), the only one that cannot be expressed in the Horn fragment is the requirement that S be a successor. *If, however, we augment our logic with a successor relation*, we obtain the desired result.

Let us define precisely what we mean: We say that a graph-theoretic property $\mathcal{G}$ *is expressible in Horn existential second-order logic with successor* if there is a Horn existential second-order expression ϕ with two binary relational symbols G and S, such that the following is true: For any model M appropriate for ϕ *such that* S^M *is a linear order on the nodes of* G^M, $M \models \phi$ if and only if $G^M \in \mathcal{G}$.

Theorem 8.4: The class of all graph-theoretic properties expressible in Horn existential second-order logic with successor is precisely **P**.

Proof: One direction was proven as Theorem 5.9. For the other direction, given a deterministic Turing machine M deciding graph-theoretic property $\mathcal{G}$ within time n^k, we shall construct an expression in Horn existential second-order logic that expresses $\mathcal{G}$ (assuming, of course, that S is a successor). The construction is identical to that of the previous proof, except a little simpler. The constituents of P are now just the T_σ's, since S is now a part of our basic vocabulary. More importantly, there is no C_0 or C_1, since the machine is deterministic. *As result, the expression produced is Horn.* The proof is complete. □

Recall now the special case of SAT with Horn clauses, shown polynomial in Theorem 4.2.

Corollary: HORNSAT is **P**-complete.

Proof: The problem is in **P** by Theorem 4.2. And we know from the proof of Theorem 5.9 that any problem of the form ϕ-GRAPHS, where ϕ is a Horn expression in existential second-order logic, can be reduced to HORNSAT. But Theorem 8.4 says that this accounts for all problems in **P**. □

8.4 NOTES, REFERENCES, AND PROBLEMS

8.4.1 There are many different kinds of reductions; our logarithmic-space reduction is about the weakest kind that has been proposed (and is therefore more useful and convincing as evidence of difficulty); but see Problem 16.4.4 for even weaker reductions, useful for **L** and below. Surprisingly, it is all we need in order to develop the many completeness results in this book—besides, demonstrating that a reduction can be carried out in logarithmic space is usually very easy. Traditionally **NP**-completeness is defined in terms of *polynomial-time many-one reduction* (also called *polynomial transformation*, or *Karp reduction*); and logarithmic-space reductions are only used for **P** and below. It is open whether **NP**-complete problems under the two definitions coincide.

A polynomial-time *Turing reduction* or *Cook reduction* is best explained in terms of oracle machines (see the definition in Section 14.3): Language L Cook-reduces to L' if and only if there is a polynomial-time oracle machine $M^?$ such that $M^{L'}$ decides L. In other words, polynomially many queries of the type "$x \in L'$?" are allowed (and not just one and in the end as with Karp reductions). Polynomial Turing reductions appear to be much stronger (see Section 17.1).

There is an intermediate form of reduction called *polynomial-time truth-table reduction*. In this reduction we can ask several "$x \in L'$?" queries *but all must be asked before any of them is answered.* That is, we obtain the final answer as a Boolean function of the answers (hence the name). For interesting results concerning the four kinds of reductions see

- R. E. Ladner, N. A. Lynch, and A. L. Selman "A comparison of polynomial time reducibilities," *Theor. Comp. Sci., 1*, pp. 103–124, 1975.

But there are many other kinds of reductions: For *nondeterministic reductions* see Problem 10.4.2, and for *randomized reductions* see Section 18.2. In fact, studying the behavior of different kinds of reductions (in their broader sense that includes oracles) and making fine distinctions between them comprises a major part of the research activity in the area known as *structural complexity* (whose annual conference is referenced many times in this book). For a nice exposition of that point of view of complexity see

- J. L. Balcázar, J. Diaz, J. Gabarró *Structural Complexity, vols. I and II*, Springer-Verlag, Berlin, 1988.

Obviously, this direction is rather orthogonal to our concerns here.

8.4.2 Problem: A *linear-time reduction* R must complete its output $R(x)$ in $\mathcal{O}(|x|)$ steps. Prove that there are *no* **P**-complete problems under linear-time reductions. (Such a problem would be in $\mathbf{TIME}(n^k)$ for some fixed $k > 0$.)

8.4.3 Problem: Prove Proposition 8.3, namely that the classes **P**, **NP**, **coNP**, **L**, **NL**, **PSPACE**, and **EXP** are closed under reductions. Is $\mathbf{TIME}(n^2)$ closed under reductions?

8.4.4 Generic complete problems. Show that all languages in $\mathbf{TIME}(f(n))$ reduce to $\{M; x : M \text{ accepts } x \text{ in } f(|x|) \text{ steps}\}$, where $f(n) > n$ is a proper complexity

function. Is this language in $\mathbf{TIME}(f(n))$?

Repeat for nondeterministic classes, and for space complexity classes.

8.4.5 If $\mathcal{C}$ is a complexity class, a language L is called *$\mathcal{C}$-hard* if all languages in $\mathcal{C}$ reduce to L *but L is not known to be in $\mathcal{C}$*. $\mathcal{C}$-hardness implies that L cannot be in any weaker class closed under reductions, unless $\mathcal{C}$ is a subset of that class. But of course L could have much higher complexity than any language in $\mathcal{C}$, and thus it fails to capture the class. For example, many languages decidable in exponential time or worse are trivially **NP**-hard, but they certainly are not as faithful representatives of **NP** as the **NP**-complete problems. We shall not need this concept in this book.

8.4.6 Cook's theorem is of course due to Stephen Cook:

- S. A. Cook "The complexity of theorem-proving procedures," *Proceedings of the 3rd IEEE Symp. on the Foundations of Computer Science*, pp. 151–158, 1971.

A subsequent paper by Richard Karp pointed out the true wealth of **NP**-complete problems (many of these results are proved in the next chapter), and therefore the significance of **NP**-completeness:

- R. M. Karp "Reducibility among combinatorial problems," pp. 85–103 in *Complexity of Computer Computations*, edited by J. W. Thatcher and R. E. Miller, Plenum Press, New York, 1972.

Independently, Leonid Levin showed that several combinatorial problems are "universal for exhaustive search," a concept easily identified with **NP**-completeness (Cook's theorem is sometimes referred to as *the Cook-Levin theorem*).

- L. A. Levin "Universal sorting problems," *Problems of Information Transmission, 9*, pp. 265–266, 1973.

The **P**-completeness of the CIRCUIT VALUE problem (Theorem 8.1) was first pointed out in:

- R. E. Ladner "The circuit value problem is log space complete for P," *SIGACT News, 7*, 1, pp. 18–20, 1975.

8.4.7 Problem: (a) Prove that CIRCUIT VALUE remains **P**-complete even if the circuit is planar. (Show how wires can cross with no harm to the computed value.)

(b) Show that CIRCUIT VALUE can be solved in logarithmic space if the circuit is *both* planar and monotone. (The two parts are from

- L. M. Goldschlager "The monotone and planar circuit value problems are complete for P," *SIGACT News 9*, pp. 25–29, 1977, and
- P. W. Dymond, S. A. Cook "Complexity theory of parallel time and hardware," *Information and Comput., 80*, pp. 205–226, 1989

respectively. So, if your solution of Part (a) did not use NOT gates, maybe you want to check it again...)

8.4.8 Problem: (a) Define a *coding* κ to be a mapping from Σ to Σ, (not necessarily one-to-one). If $x = \sigma_1 \ldots \sigma_n \in \Sigma^*$, we define $\kappa(x) = \kappa(\sigma_1) \ldots \kappa(\sigma_n)$. Finally, if $L \subseteq \Sigma^*$ is a language, define $\kappa(L) = \{\kappa(x) : x \in L\}$. Show that **NP** is closed under codings.

In contrast, **P** is probably *not* closed under codings, but of course, in view of (a), we cannot prove this without establishing that $\mathbf{P} \neq \mathbf{NP}$. Here is the best we can do:

(b) Show that **P** is closed under codings if and only if $\mathbf{P} = \mathbf{NP}$. (Use SAT.)

8.4.9 Problem: Let $f(n)$ be a function from integers to integers. An *$f(n)$-prover* is an algorithm which, given any valid expression in first-order logic that has a proof in the axiom system in Figure 5.4 of length ℓ, will find this proof in time $f(\ell)$. If the expression is not valid, the algorithm may either report so, or diverge (so the undecidability of validity is not contradicted).

(a) Show that there is a k^n-prover, for some $k > 1$.

In a letter to John von Neumann in 1956, Kurt Gödel hypothesized that an n^k-prover exists, for some $k \geq 1$. For a full translation of this remarkable text, as well as for a discussion of modern-day complexity theory with many interesting historical references, see

- M. Sipser "The history and status of the P versus NP problem," *Proc. of the 24th Annual ACM Symposium on the Theory of Computing*, pp. 603–618, 1992.

Problem: Show that there is an n^k-prover, for some $k \geq 1$, if and only if $\mathbf{P} = \mathbf{NP}$.

8.4.10 Fagin's theorem 8.3 is from

- R. Fagin "Generalized first-order spectra and polynomial-time recognizable sets," pp. 43–73 in *Complexity of Computation*, edited by R. M. Karp, SIAM-AMS Proceedings, vol. 7, 1974.

Theorem 8.4 was implicit independently in

- N. Immerman "Relational queries computable in polynomial time," *Information and Control, 68*, pp. 86–104, 1986;
- M. Y. Vardi "The complexity of relational query languages," *Proceedings of the 14th ACM Symp. on the Theory of Computing*, pp. 137–146, 1982; and
- C. H. Papadimitriou "A note on the expressive power of PROLOG," *Bull. of the EATCS, 26*, pp. 21–23, 1985.

The latter paper emphasizes an interesting interpretation of Theorem 8.3 in terms of the logic programming language PROLOG: Functionless PROLOG programs can decide precisely the languages in **P**. The current statement of Theorem 8.4 is based on

- E. Grädel "The expressive power of second-order Horn logic," *Proc. 8th Symp. on Theor. Aspects of Comp. Sci.*, vol. 480 of Lecture Notes in Computer Science, pp. 466–477, 1991.

8.4.11 Problem: Give an expression in first-order logic describing the successor function S in the proof of Fagin's theorem (Theorem 8.3). (Define a Hamilton path P as in Example 5.12, only without requiring that it be a subgraph of G, and then define a new relation S that omits all transitive edges from P.)

8.4.12 Problem: We can state Fagin's theorem without redefining **NP** as a class of sets of graphs, as follows: **NP** is precisely the class of all languages that are reducible

to a graph-theoretic property which is expressible in existential second-order logic.
(a) Prove this version of Fagin's theorem. (Encode strings as graphs.)
(b) State and prove a similar version of Theorem 8.4.

8.4.13 Problem: Show that **NP** is precisely the set of all graph-theoretic properties which can be expressed in fixpoint logic with successor (recall Problem 5.9.14).

8.4.14 Problem: Sketch a direct proof of Cook's theorem from Fagin's theorem.

8.4.15 It turns out that any graph property ϕ expressible in Horn existential second-order logic obeys a powerful *zero-one law*: If all graphs on n nodes are equiprobable, then the probability that a graph with n nodes satisfies ϕ is either asymptotically zero, or asymptotically one as n goes to infinity; see

- P. Kolaitis and M. Vardi "0 – 1 laws and decision problems for fragments of second-order logic," *Proc. 3rd IEEE Symp. on Logic In Comp. Sci,* pp. 2–11, 1988.

Problem: Based on this result, show that there are trivial properties of graphs, such as the property of having an even number of edges, which are not expressible in Horn existential second-order logic without successor. (What is the probability that a graph has an even number of edges?)